I Claim:

1) A method of scheduling *function calls* in a software program in a dynamically reconfigurable computing system which includes an embedded processor and a finite number of *reconfigurable logic partitions* which are each programmed by a set of configuration bits dynamically loaded into the system's *configuration memory*, the method comprising the steps of:

a) processing each *function call* identified within the software program into both a hard implementation for *hard execution* in said *reconfigurable logic partitions* wherein a set of configuration bits associated with said each *function call* is generated and into a soft implementation for *soft execution* in said embedded processor;

b) assigning each set of said configuration bits generated during said hard implementation to a *virtual partition*;

c) constructing a *call history model* including statistical data characterizing the patterns in calling sequence of each *function call* in said software program obtained from benchmark data and initially using the *call history model* upon invocation of said software program;

d) employing a hierarchy of storage devices to form a *pyramid of staging slots* for the purpose of storing and moving each *virtual partition* from its initial memory location in said hierarchy of storage devices having the longest associated access latency to said *configuration memory* on a "*time-of-need*" basis;

e) constructing a *virtual partition table* of statistical data to track and allocate said *staging slots* for the purpose of staging said *virtual partitions* within said *pyramid*, wherein prior to execution of said software program all *virtual partitions* are assigned to *staging slots* having said longest associated access latency, and wherein, while executing said software program, *virtual partitions* are moved to

staging slots within said *pyramid* having either shorter or longer access latencies on said "*time-of-need*" basis dependent on said statistical data within said *virtual partition table*.

f) upon the invocation of a current *function call* while executing the software program, activating hard implementation of said current *function call* if its associated *virtual partition* is located within said *pyramid* in a position ready for activation, otherwise activating soft implementation of said *function call*.

g) during activation of the current *function call*, utilizing said *call history model* to determine probable next *function calls* to follow the current *function call*;

h) during activation of the current *function call*, simultaneously initiating *dynamic scheduling* tasks to facilitate said staging of said *virtual partitions* through the hierarchy of storage devices dependent on said patterns in call sequence indicated in said *call history model*;

i) upon the completion of activation of the current *function call*, dynamically updating said statistical data associated with the current *function call* in said *call history model* using statistical data obtained during the execution of the current *function call*;

j) repeating steps (f) to (i) for said each *function call* until termination of execution of said software program.

2) The method as described in Claim 1 wherein said step of processing each *function call* identified within the software program into both a hard implementation and a soft implementation comprises the steps of:

inserting within said software program a first pair of code statements for identifying blocks of code corresponding to each *function call*, bounding each said block of code with a start statement in the front and an end statement at the end; further inserting within each said block of code a second pair of code statements

identifying sub-blocks of code within each of said block of code targeted to be executed in said *reconfigurable logic partitions*, each sub-block of code having an associated function performable within said reconfigurable logic partition;

transcribing said associated function of said each sub-block of code to generate each of said sets of configuration bits which, when loaded into the *configuration memory* of said reconfigurable computing system, causes one or more of the said *reconfigurable logic partitions* to perform said associated function of said each sub-block of code;

compiling a first code corresponding to said soft implementation and a second code corresponding to said hard implementation, wherein said first code is executed in said embedded processor, and wherein said second code is executed such that said configuration bits corresponding to said each sub-block of code is transferred to said *reconfigurable logic partitions* for execution;

assembling said first and second codes and configuration bits to form an *executable code* for said software program.

3) The method of scheduling *function calls* as described in Claim 1 further comprising the step of including within said *virtual partition table* a plurality of entries corresponding to each *function call* and associated *virtual partition*, each entry including:

a *locator* pointer to a linked list of *address words* each for locating said associated *virtual partition* within said *pyramid*;

a *tenure* value entry showing a desired *rank* for said associated *virtual partitions*;

call-id entry for linking back to said associated *virtual partition's function call*;

in-demand entry for tracking anticipated demand for said *virtual partition*;

a *time window* entry providing the upper and lower bounds for said "*time-of-need*";

a *time-to-enter* entry providing the earliest time of activation of said *virtual partition*;

a *time-to-leave* entry providing the latest time of deactivation of said *virtual partition*;

a *prediction* entry which sums up a composite *probability* of being activated within said upper and lower bounds of said *time window*; and

a *opportunity* entry which sums up a composite *payback* value that can be anticipated from executing said *virtual partition* in said *hard execution*.

4) The method of scheduling *function calls* as described in Claim 3 further including the step of storing along with each of said *address words*:

a *rank* and *slot* entry indicating the location of said *virtual partition* within a given storage device of said *pyramid*; and

access control flags defining memory access privileges of said location of said *virtual partition* within said hierarchy of storage devices.

5) The method of scheduling *function calls* as described in Claim 1 wherein said step of constructing said statistical *call history model* further comprises creating a *function call table* including a plurality of entries corresponding to each *function call*, each *function call* entry including:

a pointer to a linked list of probable *next-calls* entries;

a *speed-up* factor corresponding to a *performance gain* factor of said each *function call* executed in *hard execution*;

a *hard-duration* time corresponding to the length of time to execute said each *function call* in *hard execution*;

a *macro-set* pointer which points to a linked list of *virtual partition* identifiers associated with said *function call*.

6) The method of scheduling *function calls* as described in Claim 5 further comprising the step of including within each probable *next-call* entry:

a probable *next-call* id;

a list pointer to a next probable *next-call* in said list of probable *next-calls*;

a *probability* factor of said probable *next-call*;

a *time-gap* corresponding to the separation in time between two calls in succession.

7) The method as described in Claim 1 wherein said step of initiating *dynamic scheduling* tasks includes the step of performing a *demand look-ahead* task comprising the steps of:

recursively traversing *next-calls* lists including a list of said probable next *function calls* in said *call history model* a predetermined number of (*k*) times to establish a *tree of next-calls* that is to follow said current *function call*, wherein *k* is defined as a *look-ahead depth*;

and predicting said "*time-of-need*", a *probability* factor, and an *expected payback* factor, for each of said *next-calls* in said *tree*.

8) The method as described in Claim 7 wherein, upon completion of said *demand look-ahead* task, said step of initiating *dynamic scheduling* tasks further including the step of prioritizing said *virtual partitions* associated with each of said *next-calls* in said *tree* into three orderings including a *temporal order* based on said

"*time-of-need*", a *probabilistic order* based on said *probability* factor, and *an opportunistic order* based on *said expected payback* factor.

9) The method as described in Claim 8 wherein, upon completion of said prioritizing said *virtual partitions*, said step of initiating *dynamic scheduling* tasks further including the step of performing a *tenure management* task comprising the steps of:

determining, a *tenure* value of said each of said *virtual partition* associated with *function calls* included in said *next-calls tree*, said *tenure* value corresponding to a desired *staging slot* position of said each *virtual partition* within said *pyramid*, said desired *staging slot* position being dependent on a "*just-in-time*" principle based on said associated latency of said *staging slot* position;

wherein said *tenure management* task establishes a *tenure* value that is "*just-in-time*" with respect to said "*time-of-need*".

10) The method as described in Claim 9 wherein, upon completion of said *tenure management* task, said step of initiating *dynamic scheduling* tasks further including the step of performing a *stage de-queuing* task comprising the steps of:

freeing up sufficient ones of said *staging slots* within said *pyramid* to accommodate a number of *slots* needed as the result of a change in *tenure* value determined during said step of performing said *tenure management* task.

11) The method as described in Claim 10 wherein, upon completion of said *stage de-queuing* task, said step of initiating *dynamic scheduling* tasks further including the step of performing a *stage en-queuing* task comprising the steps of:

allocating free *staging slots* within said *pyramid* to said each *virtual partition* when its *rank* is lower than its *tenure* value; and

moving by copying said *virtual partitions* into said free *staging slots*.

12) The method of scheduling *function calls* as described in Claim 1 wherein said hierarchy of storage devices include hard disk, system main memory, dedicated external SRAM, dedicated on-chip buffer memory, and on-chip *Configuration Cache*.

13) The method as described in Claim 1 further comprising the step of storing said each set of configuration bits assigned to each *virtual partition* into more than one *staging slot*, and chaining together said more than one *staging slots* with *address words*.

14) The method as described in Claim 13 further comprising the step of managing the storing of *virtual partitions* within said *staging slots* using flag fields associated with said address words.

15) The method as described in Claim 14 wherein said flag fields include:

a *valid* field for indicating the validity of said set of configuration bits copied into said *staging slot*;

a *lock* field for prohibiting freeing-up of said *staging slot*;

a *park* field for indicating a given level within said *pyramid* in which said method of scheduling no longer controls movement of *virtual partitions* within said *pyramid of staging slots* and instead said movement is controlled by another computing system control mechanism; and

a *persistent* field for indicating said *staging slot* should never be reassigned a new *virtual partition*.

16) The method as described in Claim 7 wherein said *demand look-ahead* task further comprises the step of determining a composite *probability* factor for a sequence of more than one probable next-call.

17) The method as described in Claim 11 further comprising the steps of:

incremental tasking of said *dynamic scheduling* tasks such that said *dynamic scheduling* tasks are recursively performed on a rank-by-rank basis; and

interrupting said incremental tasking any time a new *function call* is activated wherein the most critical iterations of said *dynamic scheduling* tasks are accomplished for scheduling of said next *function call*.

18) The method as described in Claim 11 wherein a global fine tuning process is employed to automatically adjust "greediness" of computational algorithms used to perform said dynamic scheduling tasks.

19) The method as described in Claim 18 wherein said computational algorithms are formulated with simple linear computational relationships which can be weighted by a reduction fraction (*f*) and a balance of said reduction fraction (1-*f*), based on recent and historical statistical data in said function call history model.

20) The method as described in Claim 1 further comprising the step of scheduling said *function calls* by performing a *training mode*, said training mode comprising the steps of:

by-passing said *call history model* based on benchmark data;

upon invocation of said each *function call* during an initial software program run-time, activating both said *hard implementation* and said *soft implementation* of said each *function call*;

constructing a *call history model* with statistical data logged relating to said *hard implementation* and *soft implementation* of said each *function call* during said initial run-time; and

upon subsequent invocations of said each *function call*, dynamically updating said *call history model* constructed during said initial run-time, wherein said *call history model* is constructed on-the-fly during software program run-time.